Thierry Stein
Stage de 3ème année d'IUP MAI

Génération d'un serveur de fonctions

Stage effectué au Laboratoire d'Astrophysique de Grenoble
du 5 avril au 31 aout 2004
sous la direction de Messieurs Gérard Zins et Guillaume Mella

Remerciements

Avant toute chose, je tiens à remercier Gérard Zins de m'avoir confié ce stage. J'ai particulièrement apprécié sa bonne humeur permanente, qui ne nuit pas pour autant à sa rigueur et à sa compétence. Il appartient à cette catégorie rare de personne que toute entreprise ou laboratoire souhaite posséder.

Je remercie chaleureusement Guillaume Mella de m'avoir encadré pendant toute la durée de mon stage. Ses conseils, son soutien et sa culture informatique m'ont été d'un grand secours lorsque je me trouvais dans une impasse. Je pense de plus que sa fréquentation a été formidablement enrichissante.

Je remercie sincèrement les membres du JMMC : Gilles Duvert, Laurence Gluck, Sylvain Lafrasse, et Jean Berezne pour leur bonne humeur et leur disponibilité. Je remercie de même Alain Chelli, le directeur du JMMC, pour m'avoir accueuilli au sein de son équipe.

Merci à Sylvain, Alfred et Sophie (j'ignore leurs noms de familles, qu'ils m'excusent). J'ai partagé avec eux mes trois premiers mois de stages dans notre grande salle commune, et leur départ a laissé un vide non comblé.

Un grand merci à Sylvain Cetre, futur membre du JMMC. La première quinzaine d'aout, lorsque la désertification gagnait l'ensemble du batiment, il était réconfortant de le trouver là, fidèle au poste. Je le remercie aussi de m'avoir fait part de cette offre de stage. Sans lui, je n'aurais probablement pas vécu cette expérience.

Table of Contents

Introduction..4
Partie 1 : Présentation..5
1- Le LAOG..6
2- Le JMMC..7
3- Mon sujet..9
Partie 2 : Développement..10
1- Spécification du sujet..11
1.1- Un exemple : le programme Gmain..11
1.2- Exigences fonctionnelles..12
2- SWIG..13
2.1- Présentation..13
2.2- Exemple d'utilisation..14
2.3- Le fichier interface..15
3- XML et XSL..16
3.1- Un petit mot sur XML et XSL..16
3.2- Transformation au format XML..17
3.3- Utilisation de XSL..19
3.4- Amélioration du fichier XSL..20
4- Interfaçage avec Python..21
4.1- Le langage Python..21
4.2- SWIG et Python..21
5- Fonctionnalités supplémentaires..23
5.1- Fonctions en python..23
5.2- Script Python..24
5.3- Génération du Makefile..25
6- Exemple d'utilisation..27
Partie 3 : Bilan..29
1- Diagramme 30
2- Problèmes rencontrés..31
2.1- Plein de choses nouvelles..31
2.2- Gestion des types..32
3- Perspectives..33
4- Giagramme de Gantt..34
Conclusion..35
Annexes..36

Introduction

Dans le cadre de ma troisième année d'étude d'IUP Mathématiques Appliquées et Industrielles, je devais effectuer un stage d'une durée de seize semaines minimum. Ce stage s'est déroulé du 10 avril au 31 août 2004 au Laboratoire d'AstrOphysique de Grenoble (LAOG), parmi l'équipe du centre Jean Marie Mariotti (JMMC ou Jean Marie Mariotti Center).

Le sujet de mon stage était la conception et la réalisation d'un logiciel de génération de serveur de fonctions. Dans la première partie de ce rapport, j'expliquerai plus en détail la nature exacte de ce qui m'était demandé. Puis je présenterai le travail réalisé, ainsi que l'ensemble des moyens mis en oeuvre pour aboutir au résultat. La troisième partie de ce document contiendra un bilan technique de mon stage, en explicitant notamment les problèmes rencontrés. Je terminerai par une note personnelle sur l'expérience que m'ont apportée ces quatre mois.

Avant de commencer, je tiens à préciser quelles étaient mes motivations en débutant ce stage. Bien qu'étudiant à la fois en mathématiques et en informatique, je n'avais qu'une faible connaissance du monde des ordinateurs et des programmeurs. Lorsque je suis entré à l'IUP, je savais tout juste mettre en marche un PC (et encore, il fallait que le bouton soit bien en évidence). Je souhaitais donc enrichir ma culture informatique, et voir sur le terrain la réalité de la programmation que j'avais découverte en cours.

<u>**Partie 1 : Présentation**</u>

Bienvenue dans le monde réel
Andy et Larry Wachowski, *Matrix*

1- Le LAOG

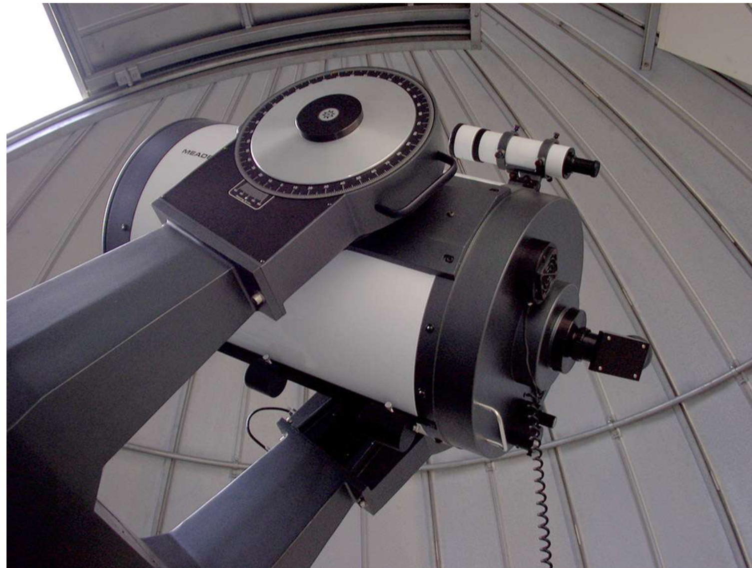
Créé en 1979, le Laboratoire d'Astrophysique de Grenoble est une unité mixte de recherche de l'Université Joseph Fourier et du Centre National pour la Recherche Scientifique. Il est à la fois un centre de recherche et un pôle de diffusion du savoir.

Les activités du laboratoire recouvrent un très large domaine de compétence :

- L'observation, bien sûr, qui est à l´origine de l'astronomie
- La modélisation numérique de phénomènes inter et intra stellaires
- La recherche théorique
- Le développement technologique.

Ceci l'amène tout naturellement à collaborer avec l'industrie et de nombreux centres de recherche locaux aussi bien qu'internationaux. Impliqué dans l'enseignements et la formation, il contribue de plus à la diffusion des connaissances auprès du grand public.

Le personnel du LAOG se compose de 30 chercheurs et enseignants chercheurs, 20 ingénieurs, techniciens et administratifs, ainsi que d'une vingtaine de visiteurs, post-doctorants, stagiaires et thésards.



De plus amples rensignements relatifs au laboratoire sont disponibles sur le site du LAOG à l'adresse :

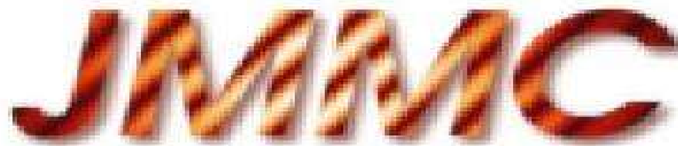
http://www-laog.obs.ujf-grenoble.fr

2- Le JMMC

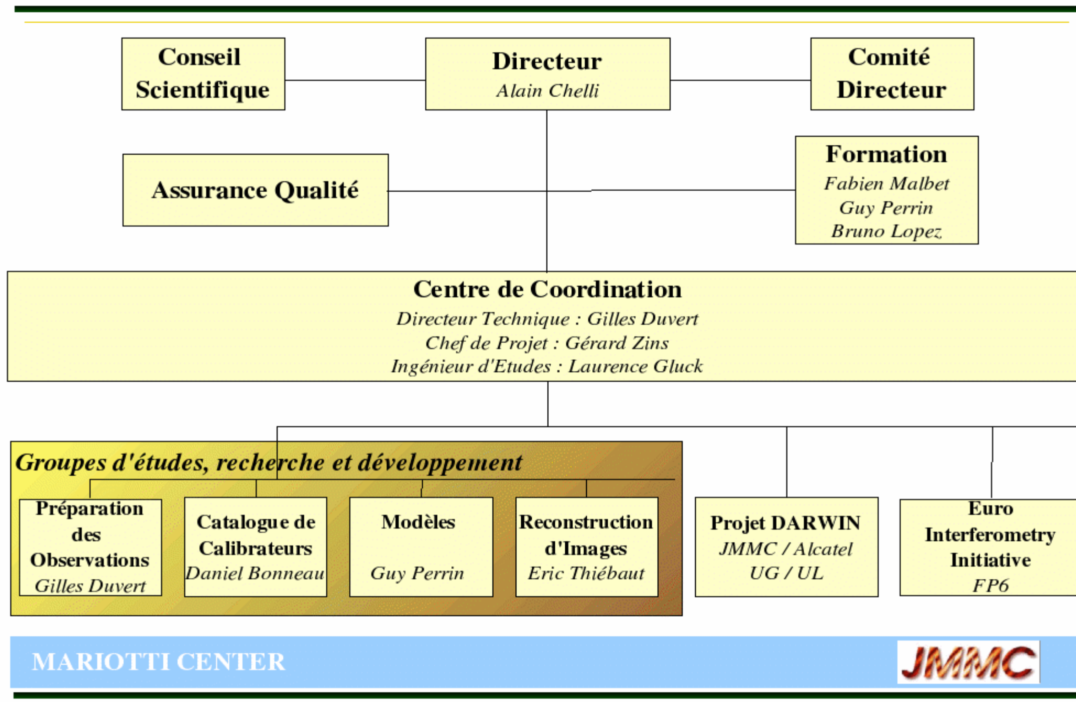
Au sein du LAOG, le Centre Jean-Marie Mariotti, ou JMMC, travaille sur la structuration et l'encadrement de développement de logiciel, concernant principalement la préparation des observations interférométriques.

L'interférométrie est une technique d'observation qui permet aux astronomes d'obtenir une résolution angulaire supérieure à celle accessible avec des téléscopes monolithiques. Cette technique se base sur le caractère ondulatoire de la lumière. Dans ce cadre, il est nécessaire de développer de nouveaux logiciels pour préparer les observations, manipuler les données, et interpréter les résultats en terme de modèle simple ou de reconstruction d'images.

Le JMMC travaille en relation avec l'Observatoire de la Côte d'Azur (OCA), ainsi qu'avec une dizaine de laboratoires français disposant d'une expertise en interférométrie.



We interfere constructively



Organigramme du JMMC

3- Mon sujet

Le sujet exact de mon stage était le conception et la réalisation d'un logiciel de génération de serveur de fonctions pour les tests unitaires et de validation de librairies C ou C++. Ce logiciel consiste à analyser le code source d'une librairie de fonctions C ou de classes C++ pour en extraire la liste des méthodes publiques. Celles ci sont ensuite rendues accessibles à l'utilisateur, qui peut appeler chacune de ces fonctions, fixer la valeur de chaque paramètre d'entrée, et voir s'afficher chaque sortie.

Le déroulement de ce logiciel peut se décomposer en deux étapes :

1- Extraire depuis les fichiers sources de la librairie les prototypes des fonctions à rendre disponible

2- Générer le code source du serveur permettant de tester chacune de ces fonctions

La première partie nécessite de savoir quels types de fichiers on peut accepter en entrée du logiciel. L'analyse peut se faire sur un fichier source ou bien sur un fichier d'entêtes, qui ne contient que les prototypes des fonctions. Il faut ensuite déterminer quel outil utiliser pour effectuer cette analyse.

La deuxième partie nécessite de mettre en place un langage d'intérrogation du serveur. L'utilisateur va appeler les fonctions par l'intermédiaire de requêtes, par exemple en saisissant au clavier le nom de la fonction et la valeur qu'il assigne à chaque paramètre. Pour interpréter ces requêtes, on a la choix entre générer un code C compilable qui pourra les comprendre, ou bien interfacer la librairie de test avec un langage interprété.

Partie 2 : Développement

Lorsque vous avez éliminé l'impossible,
ce qui reste, si improbable soit-il,
est nécessairement la vérité
Sir Arthur Conan Doyle, *Le signe des Quatre*

1- Spécification du sujet

1.1- Un exemple : le programme Gmain

Développé en 1993 au sein de l'ONERA à Chôtillon (92), le programme Gmain est ce qu'on appelle communément une «moulinette». Il est écrit en Tcl, un langage de commande interprété, principalement utilisé pour créer des interfaces graphiques.
Gmain génère un code compilable incluant toutes les fonctions documentés des fichiers C qu'on lui passe en paramètre. Son déroulement est le suivant :

1- Recherche des fichiers passés en paramètres
2- Traitement de chaque fichier :
- Lecture de la documentation pour extraire les fonctions membres
- Extraction des types d'entrée/sortie et des noms des paramètres de chaque fonction
- Ecriture dans un fichier temporaire du code pour appeler chaque fonction

3- Ecriture du «main»
4- Compilation du «main»

Gmain prend en entrée un programme C, documenté selon un format propriétaire précis (en réalité deux formats, appelé «cdoc» et «mangen»). Il utilise cette documentation pour extraire la liste des fonctions à rendre disponibles. Par exemple, supposons que l'on fournisse à Gmain le fichier C suivant :

```
/* Fichier : exemple.c */

/****************************************
* add -
* RETURNS : la somme de deux entiers
*/

int add (int a,int b) { return a+b; }
int sous (int a,int b) {return a-b; }
```

La fonction «add» est documentée au format mangen et sera interfacée par gmain. Par contre, la fonction «sous» ne sera pas disponible sur le serveur. On peut ainsi filtrer la liste des fonctions que l'on souhaite rendre disponible sur le serveur

La commande `«$ gmain.tcl exemple.c»` génère un fichier nommé par défault «main.c». Une fois celui-ci compilé, on utilise le serveur de cette manière :

```
$ main std
add -a 1 -b 2
3
sous -a 1 -b 2
invalid command name sous
```

Dans cet exemple, std est le canal de communication que l'on choisit pour utiliser le serveur, c'est à dire stdin/stdout. On constate que la fonction sous n'est pas disponible.

Ce programme m'a été fourni au début de mon stage afin de me servir d'exemple plus que de modèle. Il contient des idées intéressantes, mais est cependant limité sur plusieurs points :

- En entrée, il accepte uniquement un fichier source C.
- Il ne gère pas les types structurés.
- L'utilisation de la documentation est nécessaire et contraignante.

Une version améliorée de ce programme, extensible au C++, avait été entamée, mais le projet semble avoir été abandonné en cours de route. Heureusement pour moi, puisque dans le cas contraire ce stage n'aurait pas existé.

1.2- Exigences fonctionnelles

Après avoir étudié le programme Gmain, mon travail a consisté à rédiger un document de spécification du logiciel. Ce document devait détailler l'ensemble des fonctionnalités du futur logiciel. Il devait rester très général et ne comporter aucune proposition purement technique. Il se situe en annexe 7 de ce rapport.
Une fois celui ci rédiger, il fallait choisir entre développer une solution en interne en démarrant de zéro, ou bien utiliser des outils existants. C'est cette deuxième voie qui a été choisir.
Dans la suite de ce rapport, je vais donc vous présenter les outils utilisés afin de réaliser le logiciel, en expliquant comment j'ai agencé ces differents composants.

2- SWIG

2.1- Présentation

SWIG (Simplified Wrapper and Interface Generator) est un logiciel libre développé depuis 1995 par de nombreux informaticiens du monde entier. Son objectif principal est de permettre d'interfacer des programmes C ou C++ avec des langages interpretés (tel que Tcl ou Python). Pour réaliser cela, SWIG se comporte un peu comme un compilateur. Il prend les déclarations C et, à partir de celle ci, génère un code additionnel permettant d'accéder à ces déclarations par l'intermédiaire d'autres langages. Dans sa version actuelle (1.3), il inclut les langages Perl, Python, Tcl, Guile, Java, Ruby, Ocaml, Chicken. On dénombre de nombreuses applications de ce logiciel. Citons notamment :

- Création d'une interface interpretée pour utiliser du code C
- Débuggage interactif de programmes
- Test de librairies C
- Création d'une interface homme/machine (en utilisant Tcl/Tk par exemple)

L'ensemble des informations relatives à ce logiciel est disponible sur le site http://www.swig.org.

Quand au logiciel, il est librement téléchargeable sur le site http://sourceforge.net.

2.2- Exemple d'utilisation

Pour illustre le fonctionnement de SWIG, rien de mieux qu'un petit exemple d'utilisation. Supposons que nous disposons en entrée du code C suivant :

```
/* Fichier : exemple.c */

double variable_global = 3.0
int factoriel (int n) {
      if (n <= 1) return 1;
      else return n*factoriel(n-1);
}
```

On souhaite pouvoir accéder à la varibable global variable_global et la fonction factoriel depuis le langage Tcl. Afin que SWIG puisse créer le code additionnel, il faut lui fournir un fichier interface (contenant par convention l'extension .i). Ce fichier interface est de la forme suivante :

```
/* Fichier : exemple.i */
%module exemple
%{
%}
double variable_global;
int factoriel (int n);
```

La directive `%module` définit le nom du module qui va être généré par SWIG. Le bloc %{, %} donne la possibilité à l'utilisateur d'insérer du code supplémentaire, comme l'inclusion de fichier header, ou bien des déclarations C supplémentaires.

Une fois écrit le fichier exemple.i, on génère une librairie partagée pour utiliser le module en Tcl avec les commandes suivantes :

```
$ swig -tcl exemple.i
$ gcc -c -fpic exemple.c exemple_wrap.c
$ gcc -shared exemple.o exemple_wrap.o -o exemple.so
```

La commande swig génère un nouveau fichier appelé «exemple_wrap.c», qui contient le code additionnel permettant l'interfaçage des fonctions C avec Tcl. Les deux étapes suivantes compilent les fichiers «exemple.c» et «exemple_wrap.c», afin de créer une librairie dynamique «exemple.so». Celle ci peut ensuite être chargée lors de l'exécution d'un script Tcl, rendant utilisables les fonctions et variables déclarées dans le fichier interface.

L'utilisation des fonctions avec Tcl peut se faire de la manière suivante :

```
$ tclsh
% load ./exemple.so
% fact 4
24
%expr $variable_global + 4.5
7.5
%
```

2.3- Le fichier interface

Le fichier interface est la base de travail de SWIG. Il n'est pas indispensable, puisque SWIG pourrait fonctionner à partir d'un fichier header. Son usage est cependant recommandé car il permet d'ajouter un certain nombre de fonctionnalités au code C de départ. La forme de base du fichier interface est la suivante :

```
%module nom_module
%{
#include «fichier.h»
%}
// Déclarations de type
typedef enum {
FALSE = 0,
TRUE } Bool;
// Déclarations de variables et méthodes C ou C++
int dumbo;
Bool voler (int dumbo);
```

Le fichier interface peut contenir les informations suivantes :

- Des définitions de types ou de structures. Celle ci sont nécessaires pour pouvoir utiliser les types complexes. En l'absence de définition, SWIG traite tous les types structurés comme des pointeurs.
- Des entêtes de fonctions, définies dans le fichier source. Swig n'interfacera que les fonctions dont les prototypes se trouvent effectivement dans le fichier interface. Ceci permet de réduire le nombre de méthodes que l'on souhaite interfacer. Dans le cas des classes C++, SWIG n'interface que les méthodes publiques.
- Le code source de fonctions supplémentaires. On peut souhaiter rajouter des fonctions supplémentaires, comme par exemple des fonctions facilitant la création et l'affichage de structure.
- Des macros similaires à celles du préprocesseur C. On peut par exemple définir des constantes supplémentaires.
- L'inclusion de bibliothèques spécifiques à SWIG. SWIG dispose d'un certain nombre de librairies qui facilitent le traitement des types de données. Mentionnons notamment la librairie «cpointer.i», dont je reparlerai ultérieurement.

3- XML et XSL

3.1- Un petit mot sur XML et XSL

Le XML est un langage un peu particulier : en lui même, il ne fait absolument rien. Il a été créé pour structurer de l'information, celle ci étant encodée entre des balises. Il s'agit la de son seul point commun avec le HTML, roi du web entre 1990 et 2000.
Créé en 1999, le XML a l'avantage d'être évolutif, puisqu'aucune balise n'est prédéfinie. C'est l'utilisateur qui définit à sa convenance ses propres balises. Ce potentiel fait de ce langage le langage du futur, le destinant à devenir le standard pour tout se qui concerne la manipulation et la transmission de données, tout du moins pour le début de ce troisième millénaire.
Un fichier XML ressemble couramment à ceci :

```
<?xml version= «1.0»?>
<demoXML>
<message>Voici du XML</message>
</demoXML>
```

Un navigateur web lisant ce fichier l'affichera tel quel.

```
<?xml version= «1.0»?>
<demoXML>
<message>Voici du XML</message>
</demoXML>
```

Il faut lui fournir des informations supplémentaires pour qu'il puisse le comprendre. C'est à ce niveau qu'intervient le langage XSL.
XSL est le langage de feuille de style du XML. Cela signifie qu'il reprend des données XML et produit la présentation ou l'affichage de celle ci selon les souhaits de son créateur.
En reprenant notre exemple précedant, on peut lui joindre le fichier XSL suivant :

```
<?xml version= «1.0»?>
<xsl:stylesheet xmlns:xsl= «http://www.w3.org/TR/WD-xsl»>
<xsl:template match= «/»>
<html>
<body>
<xsl:value-of select= «demoXML/message»/>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

On associe le fichier XML et le fichier XSL en rajoutant dans le fichier XML la ligne suivante:

<?xml-stylesheet href= «fichierxsl.xsl»?>

Le résultat dans le navigateur devient :

Voici du XML

Ceci n'est pour l'instant pas très impressionnant. Cependant, le langage de feuille de style XSL se révèle très riche et permet, avec un minimum de pratique, d'afficher du XML sous la forme que l'on souhaite, mais aussi de sélectionner une partie des élements XML, de les trier ou encore de les filtrer.

3.2- Transformation au format XML

Le logiciel SWIG, présenté dans la section précedante, connaît un grand nombre de langages compilés ou interpretés avec lesquels on peut interfacer des fonctions C. Mais il sait aussi transformer un fichier source C en un fichier XML.
Supposons que l'on dispose du fichier header suivant :

```
/* Fichier : exemple.h */

double variable_global;
int factoriel (int n);
```

On peut convertir ce fichier au format XML avec la commande suivante :

```
$ swig -xml -module nom_module test.h
```

En règle générale, SWIG travaille sur un fichier contenant comme première ligne la déclaration du nom du module. Cependant, avec l'option -module, on peut contourner cette contrainte, et le faire travailler sur n'importe quel fichier. L'option -xml indique naturellement que l'on souhaite obtenir un fichier XML en sortie. Ainsi, cette ligne de commande génère un fichier nommé par défaut «test_wrap.xml».

Voici le contenu du fichier «test_wrap.xml» :


```
<?xml version="1.0" ?>
<top id="1" addr="4023da88" >
    <attributelist id="2" addr="4023da88" >
        <attribute name="outfile" value="exemple_wrap.xml" id="3" addr="4023eb08" />
        <attribute name="name" value="exemple" id="4" addr="4023eb08" />
        <attribute name="module" value="exemple" id="5" addr="4023e938" />
        <attribute name="infile" value="exemple.h" id="6" addr="4023eb08" />
        <attribute name="outfile_h" value="exemple_wrap.h" id="7" addr="4023eb08" />
    </attributelist >

    <include id="8" addr="4023d128" >

   <!-- Déclarations de typemaps spécifiques à SWIG, -->
   <!-- mais sans intérêt pour nous.                 -->

    </include >

<include id="131" addr="4023e6b8" >
        <attributelist id="132" addr="4023e6b8" >
            <attribute name="name" value="exemple.h" id="133" addr="4023eb08" />
        </attributelist >

        <cdecl id="134" addr="4023e738" >
            <attributelist id="135" addr="4023e738" >
            <attribute name="sym_name" value="variable_global" id="136" addr="4023eb08" />
            <attribute name="name" value="variable_global" id="137" addr="4023eb08" />
            <attribute name="decl" value="" id="138" addr="4023eb08" />
            <attribute name="type" value="double" id="139" addr="4023eb08" />
            <attribute name="sym_symtab" value="4023c128" id="140" addr="4023c128" />
            <attribute name="sym_overname" value="__SWIG_0" id="141" addr="4023eb08" />
            </attributelist >
        </cdecl >
        <cdecl id="142" addr="4023e8d8" >
            <attributelist id="143" addr="4023e8d8" >
               <attribute name="sym_name" value="factoriel" id="144" addr="4023eb08" />
               <attribute name="name" value="factoriel" id="145" addr="4023eb08" />
               <attribute name="decl" value="f(int)." id="146" addr="4023eb08" />
               <parmlist id="147" addr="4023e888" >
                    <parm id="148">
                        <attributelist id="149" addr="4023e888" >
                            <attribute name="name" value="n" id="150" addr="4023eb08" />
                            <attribute name="type" value="int" id="151" addr="4023eb08" />
                        </attributelist >
                    </parm >
               </parmlist >
               <attribute name="type" value="int" id="152" addr="4023eb08" />
               <attribute name="sym_symtab" value="4023c128" id="153" addr="4023c128" />
               <attribute name="sym_overname" value="__SWIG_0" id="154" addr="4023eb08" />
            </attributelist >

        </cdecl >
    </include >
</top >
```


C'est dans la deuxième partie que se trouvent les informations qui nous intéressent. Toute les informations relatives à la fonction «factoriel» se trouvent entre les balises [cdecl/attributelist]. On dispose du nom de la fonction, de son type de retour et de la liste de ses paramètres.

Dans cette première étape, je n'ai absolument rien créé de nouveau. Je me suis contenté de réecrire le fichier header de départ dans un format différent. Ce premier traitement me permet maintenant d'utiliser les fonctionnalités du XML et de son alter-ego, le langage XSL.

3.3- Utilisation de XSL

Lors de la première étape, j'ai utilisé SWIG afin de transformer un fichier header en fichier XML. Je vais maintenant lui appliquer une feuille de style XSL pour générer automatiquement le fichier interface de SWIG. Ceci va permettre d'automatiser le fonctionnement de SWIG, afin de tester ensuite des fonctions C via un langage interprété.

On peut facilement générer le fichier interface à l'aide de la feuille de style XSL suivante :

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
        <xsl:output method="text" omit-xml-declaration="yes" indent="yes"/>
        <xsl:template match = "/" >
                <xsl:text>%module </xsl:text>
                <xsl:value-of select="top/attributelist/attribute[@name='module']/@value"/>
                <xsl:text>&#xA;%{&#xA;</xsl:text>
                <xsl:for-each select="top/attributelist/attribute[@name='infile']">
                        <xsl:text>#include "</xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:text>"&#xA;</xsl:text>
                </xsl:for-each>
                <xsl:text>%}&#xA;&#xA;</xsl:text>

                <xsl:for-each select ="top/include/cdecl/attributelist/attribute[@name
='name']" >
                        <xsl:value-of select="../attribute[@name='type']/@value"/>
                        <xsl:text> </xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:if test="contains(../attribute[@name='decl']/@value,'f(')">
                                <xsl:text>(</xsl:text>
                                <xsl:for-each select=
"../parmlist/parm/attributelist/attribute[@name='type']" >
                                        <xsl:value-of select="@value"/>
                                        <xsl:text> </xsl:text>
                                        <xsl:value-of select="../attribute[@name='name']/
@value"/>
                                        <xsl:if test="not(position()=last())">
                                                <xsl:text> ,</xsl:text>
                                        </xsl:if>
                                </xsl:for-each>
                                <xsl:text>)</xsl:text>
                        </xsl:if>
                        <xsl:text>;&#xA;</xsl:text>
                </xsl:for-each>
        </xsl:template>
</xsl:stylesheet>
```

On lance ensuite la commande suivante :

```
$ xsltproc exemple.xsl exemple_wrap.xml  > exemple.i
```

Cette commande applique la feuille de style «exemple.xsl» sur le fichier «exemple_wrap.xml» et écrit le résulat dans le fichier «exemple.i». Celui ci contient alors:

```
%module exemple
%{
#include "exemple.h"
%}

double variable_global;
int factoriel(int n);
```

Une fois créé ce fichier «exemple.i», SWIG peut reprendre son déroulement normal comme décrit dans la section 1.2.

3.4- Amélioration du fichier XSL

Le fichier XSL présenté plus haut n'est pas générique, il ne couvre pas tout les cas. Le premier problème vient de SWIG lui-même. Lorsqu'il effectue la transformation du fichier header en arbre XML, il effectue un certain nombre de transformation, dont voici les plus courantes :

- «int *» est écrit sous la forme «p.int»
- «char [4]» est écrit sous la forme «a(4).char»
- «const int» est écrit sous la forme «q(const).int»

Ceci nous oblige a effectuer un traitement sur chacun des paramètres, afin de reconstruire la définition de départ. L'ensemble du code XSL commenté se trouve en annexe 2 et plus de ce rapport.

On voit que, gràce à l'utilisation combinée de SWIG, XML et XSL, on parvient à réaliser efficacement la première étape du logiciel, qui était d'extraire depuis les fichiers sources de la librairie les prototypes des fonctions à rendre disponible. On va maintenant rendre ces fonctions utilisables par l'intermédiaire du langage interprété Python.

4- Interfaçage avec Python

4.1- Le langage Python

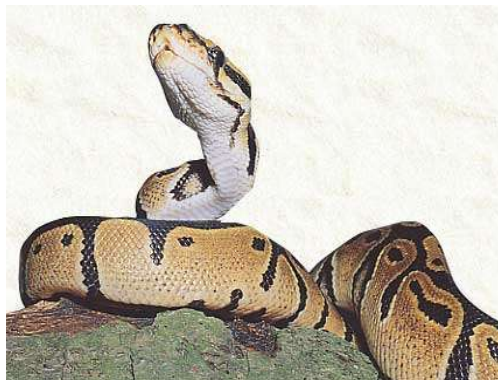
Python a été créé au début des années quatre-vingt-dix par Guido Van Rossum au *Centrum voor Wiskunde en Informatica* d'Amsterdam. C'est un langage interpreté orienté objet de haut niveau. Pour la petite histoire, son nom provient de la série télévisée Monty Python's Flying Circus (et non du serpent, comme on a tendance à le croire).
Python est un langage qui a actuellement le vent en poupe, et ce grâce à ses nombreuses qualités. Il est gratuit et portable sur les différentes variantes d'Unix aussi bien que sur MacOS et Windows. Sa syntaxe simple permet d'écrire des programmes à la fois simples et lisibles, tout en effectuant des actions très complexes. Python est extensible : on peut très facilement l'interfacer avec des librairies C existantes.
Notons par ailleurs que l'interpréteur Python est lui même écrit en C, une autre version en Java étant actuellement développée.

4.2- SWIG et Python

Pour créer un module python avec SWIG, il faut lancer la commande :

```
$ swig -python exemple.i
```

Cela génère deux fichiers distincts : un fichier source C nommé «exemple_wrap.c», et un fichier python nommé «exemple.py». Le fichier C contient le code de bas-niveau qui doit être compilé et lié avec le reste du code C pour créer le module. Le fichier python contient le code de haut-niveau. C'est lui qui sera importé lors de l'utilisation du module. Le fichier python prend le nom assigné au module. Par exemple, si le fichier «exemple.i» commence par la ligne `%module test` , SWIG génèrera un fichier nommé «test.py».



Python royal d'Australie
(effet décoratif)

L'intérêt de choisr Python pour interfacer des fonctions C est double : d'une part, la syntaxe du langage est simple, ce qui permet de tester les fonctions en ne connaissant que deux ou trois commandes python, qui sont de plus assez intuitives. De plus, le fichier «nom_module.py» permet de rajouter soi-même de nouvelles fonctionnalités au module avec un minimum de connaissances de ce langage. Ceci serait par exemple beaucoupplus difficile avec Tcl, puisque SWIG ne génère pas de fichier Tcl comme il le fait pour Python.
Si l'on reprend l'exemple de la section 3, l'appel de fonctions C en Python se fait de la manière suivante :

```
$ python
Python 2.3 (#2, Aug 31 2003, 17:27:29)
[GCC 3.3.1 (Mandrake Linux 9.2 3.3.1-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import exemple
>>> print exemple.factoriel(3)
6
>>>
```

On constate que lors de l'appel de la fonction «factoriel», on doit lui ajouter le préfixe «exemple.», c'est à dire le nom du module auquelle elle appartient. Cela peut paraitre contraignant, mais cela présente un double avantage : on peut utiliser des fonctions dont les noms seraient des mots clés spécifiques à Python. On peut aussi tester deux fonctions ayant le même nom mais définies dans deux modules différents.
On peut cependant contourner cette contrainte de cette manière :

```
>>> from exemple import *
>>> print factoriel(3)
6
```

On peut alors tester les fonctions sans écrire le préfixe. L'utilisateur a donc le choix.

5- Fonctionnalités supplémentaires

5.1- Fonctions en python

Comme dit dans la section précedente, le fichier «nom_module.py» généré par SWIG permet d'ajouter des fonctionnalités supplémentaires. Celles ci, codées en Python, s'ajoutent directement au serveur de test. Ces fonctions concernent principalement les structures.
Dans un programme C, il est fréquent que l'on utilise des types de données de la forme :

```
typedef struct {
      double re;
      double im;
} Complexe;
```

Les structures ne sont pas facile à gérer. Pour faciliter leur utilisation, mon programme écrit en Python, pour chaque structure rencontrée, les deux fonctions suivantes :

```
Complexe_lire( double re, double im)
Complexe_print( Complexe type)
```

La fonction Complexe_lire permet de créer facilement une structure, en fournissant simplement en paramètre la valeur à assigner à chaque champ.
Le fonction Complexe_print permet d'afficher le contenu de chaque champ de la structure.
Il est intéressant de coder ces fonctions en Python car cela simplifie l'affichage. En effet, la ligne `print Complexe.re` affiche directement le contenu de ce champ. On ne rencontre donc pas les problèmes relatifs au C concernants les marqueurs de type pour l'affichage, comme «%d» pour écrire un entier par exemple.
Python permet de plus d'intégrer facilement une fonction d'aide. Ainsi, on dipose à tout moment d'informations relatives au module que l'on teste et aux prototype des fonctions que l'on souhaite utiliser.

5.2- Script Python

Pour faciliter le test d'une librairie C ou dune classe C++, mon progamme écrit un fichier supplémentaire. Ce fichier, nommé par défault «test_nom_module.py», contient toutes les lignes de commandes à exécuter en Python pour tester l'ensemble des fonctions disponibles sur le serveur.
Supposons que l'on dispose du fichier header suivant :

```
/* Fichier exemple.h */
int f1(int a, int b);
char * f2(char * msg);
int f3(int a, int b);
```

On crée un module de test nommé « exemple1 ». L'utilisateur aura à sa disposition le script python suivant :

```
import exemple1

# Appel de la fonction f1
a = 0
b = 0
res = serveur.f1(a ,b)
print "Resultat de f1"
print res
# Appel de la fonction f2
msg = " "
res = serveur.f2(msg)
print "Resultat de f2"
print res
# Appel de la fonction f3
a = 0
b = 0
res = serveur.f3(a ,b)
print "Resultat de f3"
print res
```

Des valeurs par défaut sont affectées à chaque variable. L'utilisateur peut à sa convenance modifier la valeur de chaque paramètre. Il lui suffit ensuite de lancer la commande `$python test_exemple1.py` pour voir s'afficher le résultat de chaque fonction avec les valeurs qu'il a choisies.

5.3- Génération du Makefile

Afin de rendre agréable la mise en oeuvre de tout ce procédé, j'ai mis en place un script en shell (langage de commande Unix), nommé «autoSwig». Ce script génère automatiquement le Makefile contenant l'ensemble du déroulement pour mettre en place le serveur.

Ceci s'effectue à partir d'un makefile de base, contenant l'ensemble du processus à exécuter. Pour le C, le makefile de base est celui-ci :

```
XSL_SOURCE = /home/users/tstein/autoSwig/src/
MODULE_NAME =
HEADER_FILE =
SOURCE_FILE =
OBJECT_FILE =
HEADER_PATH =
LIBRARYtoPROCESS =
LIBRARY_PATH =

all: $(MODULE_NAME).so

$(HEADER_FILE).i: $(HEADER_FILE).h
             swig  -I$(HEADER_PATH)  -includeall  -ignoremissing  -module
$(MODULE_NAME) -xml -o $(HEADER_FILE).xml $(HEADER_FILE).h
             xsltproc  $(XSL_SOURCE)ginterface.xsl  $(HEADER_FILE).xml  >
$(HEADER_FILE).i

$(HEADER_FILE)_wrap.c: $(HEADER_FILE).i
              swig  $(HEADER_PATH)  -includeall  -ignoremissing  -python
$(HEADER_FILE).i
         xsltproc $(XSL_SOURCE)help_functions.xsl $(HEADER_FILE).xml >>
$(MODULE_NAME).py

$(HEADER_FILE)_wrap.o: $(HEADER_FILE)_wrap.c
               gcc  -I/usr/include/python2.3/  -c  -fpic  $(SOURCE_FILE)
$(HEADER_FILE)_wrap.c $(HEADER_PATH)

$(MODULE_NAME).so: $(HEADER_FILE)_wrap.o
         gcc -shared $(LIBRARYtoPROCESS) $(LIBRARY_PATH) $(OBJECT_FILE)
$(HEADER_FILE)_wrap.o -o _$(MODULE_NAME).so
          xsltproc  $(XSL_SOURCE)python_script.xsl  $(HEADER_FILE).xml  >
test_$(MODULE_NAME).py

clean:
              rm  -f  *~  $(HEADER_FILE).i  *_wrap.*  $(HEADER_FILE).xml
$(OBJECT_FILE) _$(MODULE_NAME).so *.py*
```

Lorsque l'utilisateur lance autoSwig, il peut ajouter de nombreuses options à cette commande. La commande `$autoSwig -help` affiche l'ensemble des options disponibles. Celle ci permettent de remplir automatiquement l'ensemble des champs du Makefile. Par exemple, la commande

```
$autoSwig -m test -h exemple -c exemple.c
```

remplit les champs MODULE_NAME, HEADER_FILE et SOURCE_FILE dans le makefile.
Voici le détail de l'ensemble des options disponibles :

`-cpp`	créer un module C++
`-m nom_module`	saisir le nom du module
`-h header`	saisir le nom du fichier header
`-c csource`	saisir le nom du ou des fichiers sources
`-I path`	saisir le chemin pour inclure d'autres fichiers
`-l library`	saisir le nom d'une librairie à lier au module
`-L library_path`	saisir le chemin d'accès à la librairie
`-np`	ne pas générer le script python
`-help`	afficher l'aide

L'utilisateur peut donc obtenir en sortie un makefile avec l'ensemble des champs remplis. Il lui suffit alors lancer `$ make` pour disposer du module de test souhaité.

6- Exemple d'utilisation

Pour terminer cette partie, voici un exemple complet d'utilisation de mon serveur de fonctions. L'utilisateur dispose en entrée du fichier exemple.h vu dans la section 5.2 :

```
/* Fichier exemple.h */
int f1(int a, int b);
char * f2(char * msg);
int f3(int a, int b);
```

Ces fonctions sont définies dans le fichier exemple.c suivant :

```
/*Fichier exemple.c*/
#include "exemple.h"

int f1(int a, int b){
return a-b;
}
char * f2(char *msg){
    return msg;
}
int f3( int a, int b){
    return a*3*b;
}
```

L'utilisateur souhaite créer un module de test baptisé «toto», qui va lui permettre de tester l'ensemble des fonctions définis dans «exemple.c».

```
$ autoSwig -m toto -h exemple -c exemple.c

>>>  CREATED --> Makefile
```

Il crée ensuite le module de test avec la commande make. Il peut finalement tester le module toto avec Python :

```
$python
>>>import toto
>>>a = toto.f1(2,1)
>>>print a
1
>>>c = toto.f2(«coucou»)
>>>print c
coucou
>>>
```

Bien sur, l'utilisateur aurait aussi pu utiliser le script «test_toto.py». Il peut donc au choix effectuer ses tests de deux manières :

- A l'aide du script python pour un test automatique simple
- Dans l'interpréteur python pour un test manuel

Les fonctionnalités seront adaptés dans le futur pour permettre des tests exhaustifs par exemple.

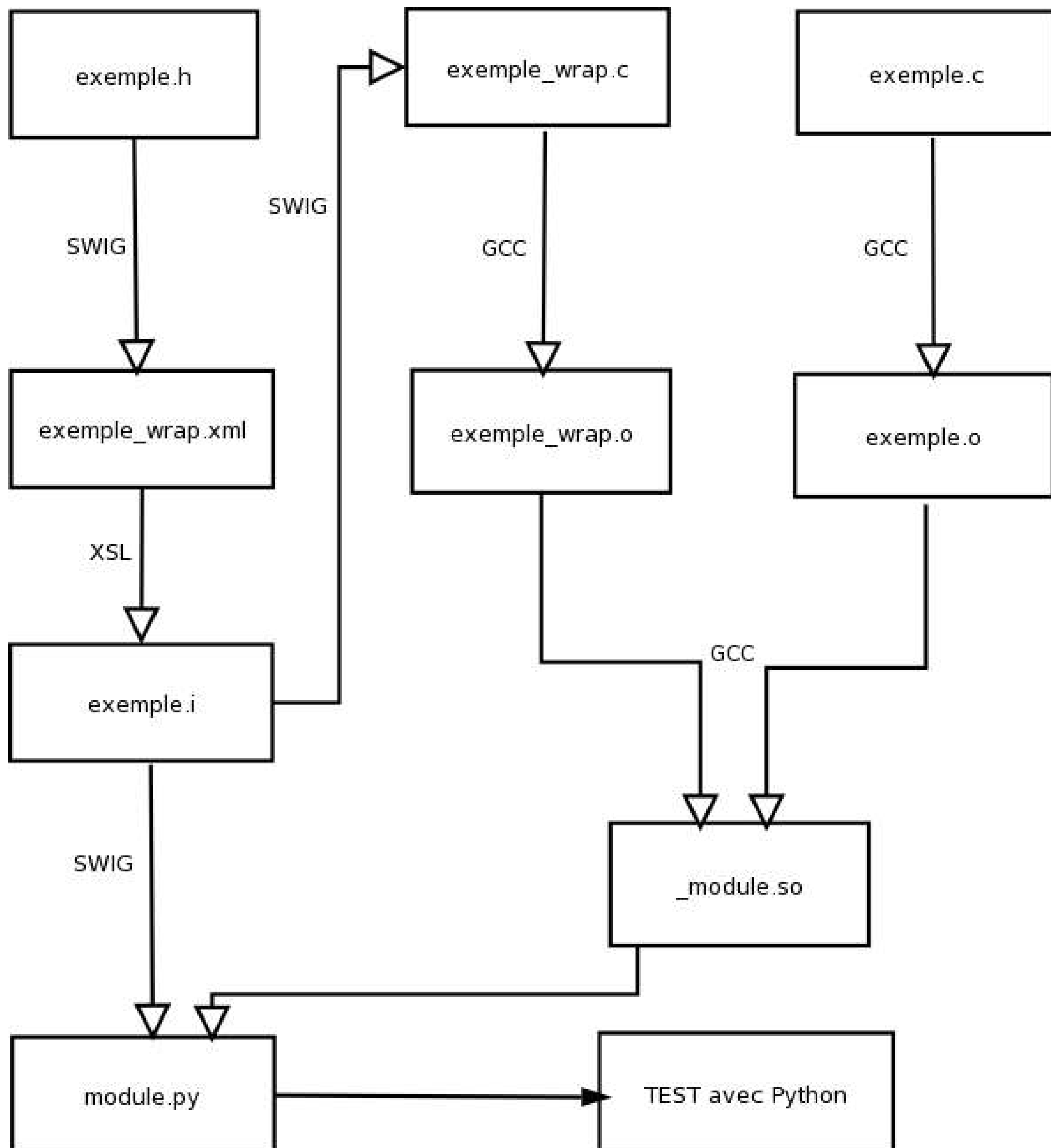
Partie 3 : Bilan

L'expérience est le nom que chacun donne à ses erreurs.
Oscar Wilde

1- Diagramme

Ce diagramme reprend l'ensemble du déroulement du logiciel. On suppose que l'utilisateur dispose en entrée d'un fichier test.c et du fichier header associé test.h. Il souhaite tester les fonctions écrites dans test.c. Pour cela, il va créer un module de test nommé «premier_test».

Voici le déroulement complet :



2- Problèmes rencontrés

2.1- Plein de choses nouvelles

Lors de ce stage, j'ai à la fois appris à utiliser un logiciel (SWIG), et découvert de nouveaux langages tels que XSL, Python, ou encore Tcl. Cela ne fut pas toujours facile, surtout pour des langages comme le XSL, qui ne ressemblait à aucun de ceux que je connaissais. La programmation en shell a été une autre découverte.
Au niveau de SWIG, j'ai du faire le tour de ses fonctionnalités afin de voir comment l'utiliser au mieux. Je ne prétends pas avoir été exhaustif dans ce travail, SWIG étant un soft disposant de nombreuses options et ressources. J'ai pour ma part choisi de m'en servir de manière assez simple, sans essayer d'utiliser toutes ses options, de peur que cela nuise à la stabilité de mon programme.
Le XSL a été une découverte intéressante. Lié au XML, il représente le futur de l'internet. C'est donc un langage au sujet duquel il est intéressant d'avoir des notions. De plus, je n'avais aucune expérience des langages balisés, ce qui constitue une autre découverte intéressante.
J'ai aussi découvert au cours de ce stage un nouvel environnement, à savoir Linux. J'avais certes une petite expérience de Unix, mais très limitée. Cela fait que les premiers jours, je me sentais perdu devant mon PC qui ne fonctionnait pas tout a fait comme avec Windows©.
Cette section aurait put être placée à un autre endroit de mon rapport, mais je tenais à la placer dans les problèmes rencontrés. Effectivement, l'apprentissage de tant de nouvelles choses fut bien une des difficultés de mon stage. Mais c'est aussi un des élements les plus bénefiques que je retiendrais de ces cinq mois.

2.2- Gestion des types

Lors du développement de mon programme, le principal problème rencontré a concerné la gestion des différents types de données. En langage C, il existe une grande variété de types disponibles, qui peuvent être simples ou structurés, scalaires ou vectoriels.

La première difficulté concernait l'utilisation des pointeurs. En python, il n'existe pas de type pointeur. Il faut donc demander à SWIG de générer plusieurs fonctions spécifiques pour gérer les pointeurs sur des types simples. Par exemple, pour une variable de type int *a, on l'initialise de cette manière en Python :

```
>>> a = new_intp()
>>> a = intp_assign(2)
```

Les fonctions new_intp() ey intp_assign() sont créées par SWIG, à condition de lui rajouter dans le fichier interface les lignes suivantes :

```
%include cpointer.i
%pointer_functions(int,intp);
```

Ceci s'applique à tout les pointeurs sur des types simples.

Un autre problème concernait les types structurés, chose qui n'était par exemple pas géré dans le programme «gmain». Pour gérer les structures avec SWIG, il faut lui écrire la définition complète dans le fichier interface. Or, les structures peuvent être déclarées de diverses manières. Par exemple :

```
typedef struct {double re; double im;} Complexe;
typedef struct complexe {double re; double im;} Complexe;
struct complexe {double re; double im;};
```

Selon que l'on se situe dans l'un ou l'autre de ces cas, SWIG ne retranscript pas la structure de la même manière dans le fichier XML.

3- Perspectives

Dans l'absolue, je n'ai pas beaucoup de perspectives à proposer par rapport à mon logiciel. Cela ne signifie pas qu'il est parfait. Son fonctionnement s'articule autour d'un pilier central : SWIG. Si l'on choisit de garder cette base, il reste peu de choses à lui ajouter.

Une piste intéressante serait l'ouverture d'un autre canal de communication. A l'heure actuel, le module de test se déroule via le canal stdin/stdout. Pour pouvoir effectuer le même travail via un canal réseau de type Tcp/Ip, il est possible d'employer la librairie XMLRPC existante pour Python. Celle ci permettrait alors de tester une librairie à distance. La librairie étant écrite en Python, cette extension pourrait se greffer autour du module dejà existant. Cependant, le recours à XMLRPC ne concerne pas uniquement mon propre module de test, mais s'inscrit dans une discussion beaucoup plus large, dans laquelle les solutions techniques sont diverses.

4- Giagramme de Gantt

ID	Task Name	Start	Finish	Duration
1	**Analyse du sujet**	**13/04/2004**	**14/05/2004**	**24d**
2	Etude de Gmain	13/04/2004	27/04/2004	11d
3	Rédaction de la spécification	28/04/2004	14/05/2004	13d
4	**Conception**	**17/05/2004**	**13/08/2004**	**65d**
5	Analyse de SWIG	17/05/2004	09/06/2004	18d
6	Ecriture des feuilles XSL	10/06/2004	06/08/2004	42d
7	Script shell et Makefile	03/08/2004	13/08/2004	9d
8	Test	21/07/2004	18/08/2004	21d
9	Rédaction du rapport	09/08/2004	27/08/2004	15d

Conclusion

Le premier enseignement que je retire de ce stage est qu'il est inutile de chercher à réinventer la roue. Dans le monde de l'informatique, de nombreux outils existent déja et sont disponibles, pour peu que l'on sache où aller les chercher. Leur usage combiné permet souvent d'aboutir à un résultat largement supérieur à ce que l'on aurait en commençant réellement de zéro. C'est exactement ce que j'ai fait tout au long de ces presque cinq mois.

De plus, j'ai appris que le développement d'un logiciel n'est pas qu'une simple affaire de codage. Une première phase d'analyse rigoureuse simplifie considérablement la suite du travail. J'avoue que lors des deux premiers mois, alors que je me contentais de spécifier formellement le futur logiciel, je n'avais pas l'impression de progresser. Mais lorsque j'ai commencé à programmer, j'ai constaté alors tout le bénefice que je retirais de cette première étape.

Je pense aussi que ces quelques mois m'ont permis de combler plusieurs lacunes dans ma culture informatique de base, ce qui était mon objectif avoué lorsque j'ai débuté ce stage. De même, le fait de fréquenter des personnes passionnées a été très enrichissant.

Finalement, je retire une très bonne expérience de stage. Grâce à ces quatre mois et demi, je me sens maintenant plus en confiance au moment d'appuyer sur le bouton «on».

Annexes

1- Script Shell pour la génération automatique du Makefile

```
#!/bin/sh
help()
{
  cat <<HELP
Manuel d'aide pour AutoSwig
Options :
-m nom_module
-h header
-c csource
-I path
-l library
-L library_path
-np : ne pas gnrer le script python
HELP
  exit 0
}

function filelen()
{
FILEEXTENDED=$1
EXT=${FILEEXTENDED##*.}

let FILELEN=${#FILEEXTENDED}-${#EXT}-1
if [ ${#EXT} -eq ${#FILEEXTENDED} ]; then
let FILELEN=${#FILEEXTENDED}
fi

FILENOTEXTENDED=${FILEEXTENDED:0:${FILELEN}}
RESULT="$FILENOTEXTENDED"
}

if test "$1" = "-cpp"
then
TEMPLATE=/home/users/tstein/autoSwig/Makefile_template/Makefile.tem
plateForTestcpp
      shift
else

TEMPLATE=/home/users/tstein/projet/Makefile_template/Makefile.templ
ateForTest
fi
grep -v "#%#" $TEMPLATE > Makefile.BAK
```

```
while [ -n "$1" ]; do
case $1 in
    -help) help;shift 1;; # appel de la fonction help
    -m)MODULE_NAME=$2;shift 2;; # ecriture du nom du module
     -h)filelen $2;len=$RESULT;HEADER_FILE=$len;shift 2;; # ecriture
du nom du fichier header
                                                  -c)SOURCE_FILE=$2;filelen
$2;len=$RESULT;OBJECT_FILE=$len.o;shift  2;;  #  ecriture  du  nom  du
fichier source
      -I)HEADER_PATH="${HEADER_PATH}  -I$2";shift  2;;  #  ecriture  du
nom du fichier header
    -l)LIBRARYtoPROCESS="${LIBRARYtoPROCESS} -l$2";shift 2;;
    -L)LIBRARY_PATH="${LIBRARY_PATH} -L$2";shift 2;;
    -np)NO_PYTHON=1;shift 1;;
    --) shift;break;; # fin des options
      -*)  echo  "erreur:  il  n'existe  pas  d'option  $1.  -help  pour
l'aide";exit 1;;
    *)  help;break;;
esac
done
sed -e "1,$ s/MODULE_NAME =/MODULE_NAME = $MODULE_NAME/g" \
    -e "1,$ s/HEADER_FILE =/HEADER_FILE = $HEADER_FILE/g" \
    -e "1,$ s/SOURCE_FILE =/SOURCE_FILE = $SOURCE_FILE/g" \
    -e "1,$ s/OBJECT_FILE =/OBJECT_FILE = $OBJECT_FILE/g" \
    -e "1,$ s|HEADER_PATH =|HEADER_PATH = $HEADER_PATH|g" \
             -e   "1,$   s/LIBRARYtoPROCESS   =/LIBRARYtoPROCESS   =
$LIBRARYtoPROCESS/g" \
    -e "1,$ s|LIBRARY_PATH =|LIBRARY_PATH = $LIBRARY_PATH|g" \
    Makefile.BAK > Makefile.tmp
rm -f Makefile.BAK
if test "$NO_PYTHON" = "1"
then
      grep -v "python_script.xsl" Makefile.tmp > Makefile
else
      mv Makefile.tmp Makefile
fi
rm -f Makefile.tmp
echo -e "\n>>>  CREATED --> Makefile\n"


#___oOo___
```

2- Code XSL pour ecrire le fochier interface

```
<?xml version="1.0"?>
<xsl:stylesheet     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
      <xsl:output      method="text"      omit-xml-declaration="yes"
indent="yes"/>

      <xsl:include href="include.xsl"/>

<!--Fontion  XSL  generant  le  fichier  interface  necessaire  a  SWIG
-->
<!--Cette fonction supporte :                                      -->
<!--  * types simples : int, float, double, char, string            -->
<!--  * pointeur sur int et double                                  -->
<!--   * typedef enum                                              -->
<!--  * typedef struct contenant des types simples ou pointeurs  -->
<!--   * tableaux 1D,2D ou nD                                      -->

      <xsl:template match = "/" >

      <!-- Ecriture du nom du module, ainsi que du ou des fichiers a
inclure -->

            <xsl:text>&#xA;%module </xsl:text>
            <xsl:value-of         select="top/attributelist/attribute
[@name='module']/@value"/>
            <xsl:text>&#xA;%{&#xA;</xsl:text>
            <xsl:for-each         select="top/attributelist/attribute
[@name='infile']">
                  <xsl:text>#include "</xsl:text>
                  <xsl:value-of select="@value"/>
                  <xsl:text>"&#xA;</xsl:text>
            </xsl:for-each>
            <xsl:text>%}&#xA;&#xA;</xsl:text>

<!-- Ces  sept  lignes  permettent  l'utilisation  de  pointeurs  sur  des
types simples -->

            <xsl:text>%include cpointer.i&#xA;</xsl:text>
            <xsl:text>%pointer_functions(int,intp);&#xA;</xsl:text>
            <xsl:text>%pointer_functions(double,doublep);
&#xA;</xsl:text>
            <xsl:text>%pointer_functions(short,shortp);
&#xA;</xsl:text>
            <xsl:text>%pointer_functions(long,longp);
&#xA;</xsl:text>
            <xsl:text>%pointer_functions(unsigned
```

```
long,unsigned_longp);&#xA;</xsl:text>
            <xsl:text>%pointer_functions
(unsignedshort,unsigned_shortp);&#xA;</xsl:text>
            <xsl:text>%pointer_functions(float,floatp);
&#xA;</xsl:text>

            <xsl:for-each         select="top/attributelist/attribute
[@name='infile']">
                  <xsl:text>%include "</xsl:text>
                  <xsl:value-of select="@value"/>
                  <xsl:text>"&#xA;</xsl:text>
            </xsl:for-each>

            <xsl:for-each                                      select
="top/include/include//cdecl/attributelist/attribute
[@name='storage' and @value='typedef']" >
                  <xsl:variable    name="Var"    select="../attribute
[@name='name']/@value"/>
                  <xsl:variable name="courant" select="current()"/>
                  <xsl:for-each       select="/top/include//attribute
[@name='type' and contains(@value,$Var)]">
                        <xsl:if test="position()=last()">
                              <xsl:call-template name="typedef">
                                    <xsl:with-param       name="noeud"
select="$courant"/>
                              </xsl:call-template>
                        </xsl:if>
                  </xsl:for-each>
            </xsl:for-each>
            <xsl:for-each
select="top/include/include//class/attributelist/attribute
[@name='name']">
                  <xsl:variable name="Var" select="@value"/>
                  <xsl:variable name="courant" select="current()"/>
                  <xsl:for-each       select="/top/include//attribute
[@name='type' and contains(@value,$Var)]">
                        <xsl:if test="position()=last()">
                              <xsl:call-template name="struct">
                                    <xsl:with-param       name="noeud"
select="$courant"/>
                              </xsl:call-template>
                        </xsl:if>
                  </xsl:for-each>
            </xsl:for-each>

      </xsl:template>
</xsl:stylesheet>
```

3- Fichier include.xsl contenant un certain nombre de fonctions pour traiter les entêtes de fonctions et les typedefs

```
<?xml version="1.0"?>
<xsl:stylesheet     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
      <xsl:output      method="text"      omit-xml-declaration="yes"
indent="yes"/>



<!-- Cette fonction sert a isoler le nom exact du type         -->
<!-- Exemple : Dans le fichier XML, un pointeur sur un entier  -->
<!--           est stocke sous la forme p.int.                 -->
<!-- Cette fonction remplace donc p.int par int *              -->

<xsl:template name="Write_type">
      <xsl:param name="Type"/>
            <xsl:choose>
            <xsl:when test="contains($Type,'f(')">
                  <xsl:choose>
                        <xsl:when test="contains($Type,').')">
                              <xsl:variable    name="inter"    select=
"concat('f(',substring-after($Type,').'))"/>
                              <xsl:call-template name="Write_type">
                                    <xsl:with-param        name="Type"
select="$inter"/>
                              </xsl:call-template>
                        </xsl:when>
                        <xsl:otherwise>
                              <xsl:call-template name="Write_type">
                                    <xsl:with-param        name="Type"
select="substring-after($Type,'f(')"/>
                              </xsl:call-template>
                        </xsl:otherwise>
                  </xsl:choose>
            </xsl:when>
            <xsl:when test="contains($Type,'...')">
                  <xsl:text>...</xsl:text>
            </xsl:when>
            <xsl:when test="starts-with($Type,'q(')">
                  <xsl:value-of    select="substring-after(substring-
before($Type,').'),'q(')"/>
                  <xsl:text> </xsl:text>
                  <xsl:value-of     select="substring-after($Type,').
')"/>
            </xsl:when>
```

```
            <xsl:when test="starts-with($Type,'a(')">
                  <xsl:variable name="nom_tab">
                        <xsl:call-template name="tab_rec">
                              <xsl:with-param           name="Tableau"
select="$Type"/>
                        </xsl:call-template>
                  </xsl:variable>
                  <xsl:call-template name="Write_type">
                        <xsl:with-param                     name="Type"
select="$nom_tab"/>
                  </xsl:call-template>

            </xsl:when>
            <xsl:when test="starts-with($Type,'p.')">
                  <xsl:call-template name="Write_type">
                        <xsl:with-param                     name="Type"
select="substring-after($Type,'p.')"/>
                  </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="$Type"/>
            </xsl:otherwise>
      </xsl:choose>
</xsl:template>

<!-- Cette fonction sert a gerer les fonctions et les variables -->
<!-- de type pointeur. Elle utilise la fonction recursive ptr   -->

<xsl:template name="pointeur">
      <xsl:param name="Type"/>
      <xsl:if test="contains($Type,'p.')">
            <xsl:variable name="interType">
                  <xsl:choose>
                        <xsl:when test="contains($Type,'f(')">
                              <xsl:value-of   select="substring-before
($Type,'(')"/>
                        </xsl:when>
                        <xsl:otherwise>
                              <xsl:value-of select="$Type"/>
                        </xsl:otherwise>
                  </xsl:choose>
            </xsl:variable>
            <xsl:call-template name="ptr">
                  <xsl:with-param name="point" select="$interType"/>
            </xsl:call-template>
      </xsl:if>

</xsl:template>
```

```
<!--  Cette  fonction  recursive  ecrit  autant  d'etoiles  que
necessaires.  La  recursivite  permet  de  gerer  des  pointeurs  de
pointeurs  -->

<xsl:template name="ptr">
      <xsl:param name="point"/>
      <xsl:if test="contains($point,'p.')">
            <xsl:text>*</xsl:text>
            <xsl:call-template name="ptr">
                  <xsl:with-param   name="point"   select="substring-
after($point,'p.')"/>
            </xsl:call-template>
      </xsl:if>
</xsl:template>

<!-- Cette fonction permet d'isoler le nom d'une variable de type
tableau. Elle trouve son utilite dans le cas de paramatre du type
suivant :
- char tableau[2][2], qui est stocke dans le fichier XML sous la a
(2).a(2).char. Pour gerer les tableau a plusieurs dimension,
elle est recursive
-->
<xsl:template name="tab_rec">
      <xsl:param name="Tableau"/>
      <xsl:choose>
            <xsl:when test="contains($Tableau,'a(')">
                  <xsl:variable   name="var"   select="substring-after
($Tableau,').')"/>
                  <xsl:call-template name="tab_rec">
                        <xsl:with-param                 name="Tableau"
select="$var"/>
                  </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="$Tableau"/>
            </xsl:otherwise>
      </xsl:choose>
</xsl:template>
```

```
<!--  Cette  fonction  sert  a  gerer  les  typedef  structures.
Elle recopie chaque champ de la structure, et genere deux fonctions
pour en faciliter l'utilisation : structure_lire et structure_print
-->

<xsl:template name="struct">
      <xsl:param name="noeud"/>
      <xsl:variable      name="type"      select="$noeud/../attribute
[@name='sym_name']/@value"/>
      <xsl:value-of     select="$noeud/../attribute[@name='storage']/
@value"/>
      <xsl:text> </xsl:text>
      <xsl:choose>
            <xsl:when
test="$noeud/../../../cdecl//attributelist/attribute[@name='name']/
@value=$type">
                  <xsl:text>typedef </xsl:text>
                  <xsl:value-of   select="../attribute[@name='type']/
@value"/>
            </xsl:when>
            <xsl:otherwise>
                  <xsl:text>struct </xsl:text>
                  <xsl:if                test="not($noeud/../attribute
[@name='storage']/@value='typedef')">
                        <xsl:value-of select="$type"/>
                  </xsl:if>
            </xsl:otherwise>
      </xsl:choose>
      <xsl:text>{&#xA;</xsl:text>
      <xsl:for-each select="$noeud/../../cdecl/attributelist">
            <xsl:for-each select="attribute[@name='type']" >
                  <xsl:call-template name="Write_type">
                        <xsl:with-param name="Type" select="@value"/>
                  </xsl:call-template>
                  <xsl:text> </xsl:text>
                  <xsl:call-template name="pointeur">
                        <xsl:with-param                     name="Type"
select="../attribute[@name='decl']/@value"/>
                  </xsl:call-template>
            </xsl:for-each>
            <xsl:value-of select="attribute[@name='name']/@value"/>
                  <xsl:if     test="contains(attribute[@name='decl']/
@value,'a(')">
                  <xsl:variable name="nom">
                        <xsl:call-template name="tab_rec">
                              <xsl:with-param           name="Tableau"
select="attribute[@name='decl']/@value"/>
                        </xsl:call-template>
                  </xsl:variable>
```

```
                <xsl:variable name="tab1">
                      <xsl:choose>
                            <xsl:when test="string-length($nom)=0">
                                  <xsl:value-of    select="attribute
[@name='decl']/@value"/>
                            </xsl:when>
                            <xsl:otherwise>
                                  <xsl:value-of   select="substring-
before(attribute[@name='decl']/@value,$nom)"/>
                            </xsl:otherwise>
                      </xsl:choose>
                </xsl:variable>
                <xsl:variable     name="tab2"     select="translate
($tab1,'a(','[')"/>
                <xsl:value-of select="translate($tab2,').',']')"/>
          </xsl:if>
          <xsl:text>;&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>} </xsl:text>
    <xsl:if            test="$noeud/../attribute[@name='storage']/
@value='typedef'  or  $noeud/../../../cdecl//attributelist/attribute
[@name='name']/@value=$type">
          <xsl:value-of select="$type"/>
    </xsl:if>
    <xsl:text>;&#xA;</xsl:text>

</xsl:template>
```

```
<!-- Cette fonction sert a ecrire la definition des typedef dans
le fichier interface. Elle support : enum, tableau 1D, 2D, ND
Elle ne traite pas les stuctures, qui sont gere plus haut   -->

<xsl:template name="typedef">
      <xsl:param name="noeud"/>
      <xsl:variable     name="type"     select="$noeud/../attribute
[@name='name']/@value"/>

      <!-- On ecarte le cas des structures, que l'on traite a part
-->
      <xsl:if  test="not(contains($noeud/../attribute[@name='type']/
@value,'struct'))">
            <xsl:text>typedef </xsl:text>
            <xsl:choose>

                  <!-- Premier cas : c'est un type enumerate -->
                  <xsl:when        test="contains($noeud/../attribute
[@name='type']/@value,'enum')">
                        <xsl:choose>
                              <xsl:when                  test="contains
($noeud/../attribute[@name='type']/@value,'$unname')">
                                    <xsl:text>enum </xsl:text>
                              </xsl:when>
                              <xsl:otherwise>
                                    <xsl:value-of
select="$noeud/../attribute[@name='type']/@value"/>
                              </xsl:otherwise>
                        </xsl:choose>
                        <xsl:text> { </xsl:text>

                        <xsl:for-each
select="$noeud/../../../enum/attributelist/attribute
[@name='name']">
                              <xsl:if test="@value=$type">
                                    <xsl:for-each
select="../../enumitem">
                                          <xsl:for-each
select="attributelist/attribute[@name='name']" >
                                                <xsl:value-of
select="@value"/>
                                                <xsl:if       test="not
(contains(../attribute[@name='enumvalue']/@value,'+1'))">
                                          <xsl:text> = </xsl:text>
                                          <xsl:value-of
select="../attribute[@name='enumvalue']/@value"/>
                              </xsl:if>
                        </xsl:for-each>
```

```
                        <xsl:if test="not(position()=last())">
                                    <xsl:text>,</xsl:text>
                        </xsl:if>
                  </xsl:for-each>
                  <xsl:text>} </xsl:text>
                  <xsl:value-of select="$type"/>
                  <xsl:text>;&#xA;</xsl:text>
            </xsl:if>
      </xsl:for-each>
</xsl:when>
                  <!-- Deuxieme cas : c'est un tableau -->
<xsl:when          test="contains($noeud/../attribute[@name='decl']/
@value,'a(')">
                        <xsl:value-of      select="$noeud/../attribute
[@name='type']/@value"/>
                        <xsl:text> </xsl:text>
                        <xsl:value-of      select="$noeud/../attribute
[@name='sym_name']/@value"/>
                        <xsl:variable  name="inter"  select="translate
($noeud/../attribute[@name='decl']/@value,'a(','[')"/>
                        <xsl:value-of     select="translate($inter,').
',']')"/>
                        <xsl:text>;&#xA;</xsl:text>
                  </xsl:when>

                  <!-- Sinon, c'est un renommage de type simple -->
                  <xsl:otherwise>
                        <xsl:value-of      select="$noeud/../attribute
[@name='type']/@value"/>
                        <xsl:text> </xsl:text>
                        <xsl:value-of      select="$noeud/../attribute
[@name='sym_name']/@value"/>
                        <xsl:text>;&#xA;</xsl:text>
                  </xsl:otherwise>

            </xsl:choose>
      </xsl:if>
</xsl:template>
```

```
<!-- Cette fonction sert à indentifier les fonctions qui renvoient
un pointeur -->
<xsl:template name="funcptr">
      <xsl:param name="decl"/>
      <xsl:choose>
            <xsl:when test="contains($decl,')')">
                  <xsl:call-template name="funcptr">
                        <xsl:with-param                    name="decl"
select="substring-after($decl,')')"/>
                  </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="translate($decl,'p.','*')"/>
            </xsl:otherwise>
      </xsl:choose>
</xsl:template>

<!-- Cette fonction reconstitue un paramètre de fonction -->
<xsl:template name="Ecrire_param">
      <xsl:param name="noeud"/>
      <!-- Ecriture du type du parametre-->
      <xsl:call-template name="Write_type">
            <xsl:with-param   name="Type"   select="$noeud/attribute
[@name='type']/@value"/>
      </xsl:call-template>
      <xsl:text> </xsl:text>
      <xsl:call-template name="pointeur">
            <xsl:with-param   name="Type"   select="$noeud/attribute
[@name='type']/@value"/>
      </xsl:call-template>

      <!-- Ecriture du nom du parametre -->
      <xsl:value-of select="$noeud/attribute[@name='name']/@value"/>

      <!-- Ecriture des dimensions dans le cas de tableau -->
      <xsl:if         test="contains($noeud/attribute[@name='type']/
@value,'(')">
            <xsl:variable name="nom">
                  <xsl:call-template name="tab_rec">
                        <xsl:with-param                 name="Tableau"
select="$noeud/attribute[@name='type']/@value"/>
                  </xsl:call-template>
            </xsl:variable>
            <xsl:variable    name="tab1"    select="substring-before
($noeud/@value,$nom)"/>
            <xsl:variable   name="tab2"   select="translate($tab1,'a
(','[')"/>
            <xsl:value-of select="translate($tab2,').',']')"/>
      </xsl:if>
</xsl:template>
```

```
<!-- Cette fonction ecrit l'entete d'une fonction -->
<xsl:template name="Entete_fonction">
      <xsl:param name="Noeud"/>
      <xsl:if    test="contains(../attribute[@name='decl']/@value,'f
(')">

      <!-- Ecriture du type de retour -->
            <xsl:call-template name="Write_type">
                  <xsl:with-param   name="Type"   select="../attribute
[@name='type']/@value"/>
            </xsl:call-template>
            <xsl:text> </xsl:text>

      <!-- Ecriture du nom de la fonction -->
            <xsl:text> </xsl:text>
            <xsl:if        test="contains(../attribute[@name='decl']/
@value,'p.')">
                  <xsl:call-template name="funcptr">
                        <xsl:with-param                    name="decl"
select="../attribute[@name='decl']/@value"/>
                  </xsl:call-template>
            </xsl:if>
            <xsl:value-of select="@value"/>
            <xsl:text>(</xsl:text>

      <!-- Ecriture des parametres de la fonction -->

            <xsl:for-each select= "../parmlist/parm/attributelist" >
                  <xsl:call-template name="Ecrire_param">
                        <xsl:with-param  name="noeud"  select="current
()"/>
                  </xsl:call-template>
                  <xsl:if test="not(position()=last())">
                        <xsl:text> ,</xsl:text>
                  </xsl:if>
            </xsl:for-each>
            <xsl:text>);</xsl:text>
      </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

4- Fichier XSL générant des fonctions d'aide en python

```
<?xml version="1.0"?>
<xsl:stylesheet     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="text" omit-xml-declaration="yes" indent="yes"/>

<xsl:include href="include.xsl"/>

<xsl:template name="Lire_structure">
      <xsl:param name="Type"/>
      <xsl:param name="noeud"/>
      <xsl:text>def </xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>_lire (</xsl:text>
      <xsl:for-each     select="$noeud/cdecl/attributelist/attribute
[@name='type']" >
            <xsl:value-of         select="../attribute[@name='name']/
@value"/>
            <xsl:if test="not(position()=last())">
                  <xsl:text> ,</xsl:text>
            </xsl:if>
      </xsl:for-each>
      <xsl:text> ) :&#xA;</xsl:text>
      <xsl:text> struct = </xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>()&#xA;     </xsl:text>
      <xsl:for-each select="$noeud/cdecl/attributelist" >
            <xsl:text>struct.</xsl:text>
            <xsl:value-of select="attribute[@name='name']/@value"/>
            <xsl:text> = </xsl:text>
            <xsl:value-of select="attribute[@name='name']/@value"/>
            <xsl:text>&#xA;  </xsl:text>
      </xsl:for-each>
      <xsl:text>return struct</xsl:text>
      <xsl:text>&#xA;</xsl:text>
</xsl:template>
```

```
<xsl:template name="Afficher_structure">
      <xsl:param name="Type"/>
      <xsl:param name="noeud"/>
      <xsl:text>def </xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>_print (</xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>):&#xA;</xsl:text>
      <xsl:text> print </xsl:text>
      <xsl:for-each select="$noeud/cdecl/attributelist" >
            <xsl:value-of select="$Type"/>
            <xsl:text>.</xsl:text>
            <xsl:value-of select="attribute[@name='name']/@value"/>
            <xsl:if test="not(position()=last())">
                  <xsl:text>, </xsl:text>
            </xsl:if>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template name="Ecrire_typedef">
      <xsl:param name="Type"/>
      <xsl:text>def </xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>_print(type):&#xA;      print type&#xA;</xsl:text>
</xsl:template>

<xsl:template name="Lire_typedef">
      <xsl:param name="Type"/>
      <xsl:text>def </xsl:text>
      <xsl:value-of select="$Type"/>
      <xsl:text>_lire(valeur):&#xA;     retour=valeur&#xA;     return
retour&#xA;</xsl:text>
</xsl:template>


<xsl:template match="/">
      <xsl:for-each    select    ="top//cdecl/attributelist/attribute
[@name='storage' and @value='typedef']" >
            <xsl:variable       name="Var"       select="../attribute
[@name='name']/@value"/>
            <xsl:variable name="courant" select="current()"/>
            <xsl:for-each       select="/top/include/cdecl//attribute
[@name='type' and contains(@value,$Var)]">
                  <xsl:if test="position()=last()">
                        <xsl:call-template name="Lire_typedef">
                              <xsl:with-param              name="Type"
select="$Var"/>
                        </xsl:call-template>
```

```
                    <xsl:call-template name="Ecrire_typedef">
                         <xsl:with-param              name="Type"
select="$Var"/>
                    </xsl:call-template>
               </xsl:if>
          </xsl:for-each>
     </xsl:for-each>

     <xsl:for-each       select="top//class/attributelist/attribute
[@name='name']">
          <xsl:variable name="Var" select="@value"/>
          <xsl:variable name="courant" select="current()"/>
          <xsl:for-each            select="/top/include//attribute
[@name='type' and contains(@value,$Var)]">
               <xsl:if test="position()=last()">
                    <xsl:call-template name="Lire_structure">
                         <xsl:with-param              name="Type"
select="$Var"/>
                         <xsl:with-param             name="noeud"
select="$courant/../.."/>
                    </xsl:call-template>
                    <xsl:call-template name="Afficher_structure">
                         <xsl:with-param              name="Type"
select="$Var"/>
                         <xsl:with-param             name="noeud"
select="$courant/../.."/>
                    </xsl:call-template>
               </xsl:if>
          </xsl:for-each>
     </xsl:for-each>
     <xsl:text>def help(nom):&#xA;</xsl:text>
     <xsl:text> if nom == "list":&#xA;</xsl:text>
     <xsl:for-each select="top/include/cdecl/attributelist">
          <xsl:if  test="contains(attribute[@name='decl']/@value,'f
(')">
               <xsl:text>      print "</xsl:text>
               <xsl:value-of      select="attribute[@name='name']/
@value"/>
               <xsl:text>"&#xA;</xsl:text>
          </xsl:if>
     </xsl:for-each>
     <xsl:for-each
select="top/include/cdecl/attributelist/attribute
[@name='sym_name']">
          <xsl:if       test="contains(../attribute[@name='decl']/
@value,'f(')">
               <xsl:text> elif nom == "</xsl:text>
               <xsl:value-of select="@value"/>
               <xsl:text>":&#xA;</xsl:text>
               <xsl:text>      print                    "prototype
```

```
is"&#xA;</xsl:text>
                  <xsl:text>       print "</xsl:text>
                  <xsl:call-template name="Entete_fonction">
                        <xsl:with-param  name="Noeud"  select="current
()"/>
                  </xsl:call-template>
                  <xsl:text>"&#xA;</xsl:text>
            </xsl:if>
      </xsl:for-each>
      <xsl:text> else:&#xA;</xsl:text>
      <xsl:text>       print      "ERROR      --      Options      not
found"&#xA;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

5- Fichier XSL générant le script d'aide en python pour des fonctions C

```
<?xml version="1.0"?>
<xsl:stylesheet     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
      <xsl:output      method="text"      omit-xml-declaration="yes"
indent="yes"/>

<xsl:include href="include.xsl"/>

<!-- Fonction XSL ecrivant un script Python   -->
<!-- facilitant le test de fonction C ou C++ -->

<xsl:template name="saisie_param">
      <xsl:param name="nom"/>
      <xsl:param name="type"/>
      <xsl:param name="noeud"/>
      <xsl:param name="module"/>
      <xsl:variable name="Type">
            <xsl:call-template name="Write_type">
                  <xsl:with-param name="Type" select="$type"/>
            </xsl:call-template>
      </xsl:variable>
      <xsl:choose>
            <xsl:when test="contains($Type,'unsigned')">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='int' or $Type='long'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='double' or $Type='float'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0.0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='string' or $Type='char'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = " "&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='void'">
                  <!-- Pas de parametre a saisir -->
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = </xsl:text>
                  <xsl:value-of select="$module"/>
                  <xsl:text>.</xsl:text>
```

```
                <xsl:value-of select="$Type"/>
                <xsl:text>_lire()&#xA;</xsl:text>
            </xsl:otherwise>
        </xsl:choose>
</xsl:template>

<xsl:template name="retour_fct">
        <xsl:param name="type"/>
        <xsl:param name="nom"/>
        <xsl:param name="module"/>
        <xsl:variable name="Type">
            <xsl:call-template name="Write_type">
                <xsl:with-param name="Type" select="$type"/>
            </xsl:call-template>
        </xsl:variable>
        <xsl:choose>
            <xsl:when    test="$Type='int'    or    $Type='long'    or
$Type='double'  or  $Type='float'  or  $Type='string'  or  $Type='char'
or contains($Type,'enum') or contains($Type,'unsigned')">
                <xsl:value-of select="$nom"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="$module"/>
                <xsl:text>.</xsl:text>
                <xsl:choose>
                    <xsl:when test="contains($Type,'struct')">
                        <xsl:value-of    select="substring-after
($Type,'struct ')"/>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:value-of select="$Type"/>
                    </xsl:otherwise>
                </xsl:choose>
                <xsl:text>_print(res)&#xA;</xsl:text>
            </xsl:otherwise>
        </xsl:choose>
</xsl:template>

<xsl:template match = "/" >
        <!-- Importation du module a tester -->
        <xsl:variable name="nom_module">
            <xsl:value-of         select="top/attributelist/attribute
[@name='module']/@value"/>
        </xsl:variable>
        <xsl:text>import </xsl:text>
        <xsl:value-of select="$nom_module"/>
        <xsl:text>&#xA;&#xA;</xsl:text>
```

```
      <!-- Ecriture des appels des fonctions -->

      <xsl:for-each                                              select
="top/include/cdecl/attributelist/attribute[@name ='sym_name']" >
            <xsl:if        test="contains(../attribute[@name='decl']/
@value,'f(')">
                  <xsl:text># Appel de la fonction </xsl:text>
                  <xsl:value-of select="@value"/>
                  <xsl:text>&#xA;</xsl:text>
      <!-- Saisie des parametres -->

            <xsl:for-each select= "../parmlist/parm/attributelist" >
                        <xsl:call-template name="saisie_param">
                              <xsl:with-param               name="nom"
select="attribute[@name='name']/@value"/>
                              <xsl:with-param              name="type"
select="attribute[@name='type']/@value"/>
                              <xsl:with-param             name="noeud"
select="current()"/>
                              <xsl:with-param            name="module"
select="$nom_module"/>
                        </xsl:call-template>
                  </xsl:for-each>

      <!-- Appel de la fonction -->

                  <xsl:if             test="not(contains(../attribute
[@name='type']/@value,'void'))">
                        <xsl:text>res = </xsl:text>
                  </xsl:if>
                  <xsl:value-of select="$nom_module"/>
                  <xsl:text>.</xsl:text>
                  <xsl:value-of select="@value"/>
                  <xsl:text>(</xsl:text>
                  <xsl:for-each                                 select=
"../parmlist/parm/attributelist" >
                        <xsl:value-of                select="attribute
[@name='name']/@value"/>
                        <xsl:if test="not(position()=last())">
                              <xsl:text> ,</xsl:text>
                        </xsl:if>
                  </xsl:for-each>
                  <xsl:text>)&#xA;</xsl:text>
      <!-- Affichage de la sortie -->
                  <xsl:if             test="not(contains(../attribute
[@name='type']/@value,'void'))">
                        <xsl:text>print "Resultat de </xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:text>"&#xA;</xsl:text>
                        <xsl:text>print </xsl:text>
```

```
                        <xsl:call-template name="retour_fct">
                              <xsl:with-param              name="type"
select="../attribute[@name='type']/@value"/>
                              <xsl:with-param
name="nom">res</xsl:with-param>
                              <xsl:with-param             name="module"
select="$nom_module"/>
                        </xsl:call-template>
                        <xsl:text>&#xA;</xsl:text>
                  </xsl:if>
            </xsl:if>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
</xsl:template>
</xsl:stylesheet>
```

6- Fichier XSL générant le script d'aide en python pour des classes C++

```
<?xml version="1.0"?>
<xsl:stylesheet     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="text" omit-xml-declaration="yes" indent="yes"/>

<xsl:include href="include.xsl"/>
<xsl:template name="saisie_param">
      <xsl:param name="nom"/>
      <xsl:param name="type"/>
      <xsl:param name="noeud"/>
      <xsl:param name="module"/>
      <xsl:variable name="Type">
            <xsl:call-template name="Write_type">
                  <xsl:with-param name="Type" select="$type"/>
            </xsl:call-template>
      </xsl:variable>
      <xsl:choose>
            <xsl:when test="contains($Type,'unsigned')">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='int' or $Type='long'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='double' or $Type='float'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = 0.0&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='string' or $Type='char'">
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = " "&#xA;</xsl:text>
            </xsl:when>
            <xsl:when test="$Type='void'">
                  <!-- Pas de parametre a saisir -->
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="$nom"/>
                  <xsl:text> = </xsl:text>
                  <xsl:value-of select="$module"/>
                  <xsl:text>.</xsl:text>
                  <xsl:value-of select="$Type"/>
                  <xsl:text>_lire()&#xA;</xsl:text>
            </xsl:otherwise>
      </xsl:choose>
</xsl:template>
```

```
<xsl:template name="retour_fct">
      <xsl:param name="type"/>
      <xsl:param name="nom"/>
      <xsl:param name="module"/>
      <xsl:variable name="Type">
            <xsl:call-template name="Write_type">
                  <xsl:with-param name="Type" select="$type"/>
            </xsl:call-template>
      </xsl:variable>
      <xsl:choose>
            <xsl:when   test="$Type='int'   or   $Type='long'   or
$Type='double' or $Type='float' or $Type='string' or $Type='char'
or contains($Type,'enum') or contains($Type,'unsigned')">
                  <xsl:value-of select="$nom"/>
            </xsl:when>
            <xsl:otherwise>
                  <xsl:value-of select="$module"/>
                  <xsl:text>.</xsl:text>
                  <xsl:choose>
                        <xsl:when test="contains($Type,'struct')">
                              <xsl:value-of   select="substring-after
($Type,'struct ')"/>
                        </xsl:when>
                        <xsl:otherwise>
                              <xsl:value-of select="$Type"/>
                        </xsl:otherwise>
                  </xsl:choose>
                  <xsl:text>_print(res)&#xA;</xsl:text>
            </xsl:otherwise>
      </xsl:choose>
</xsl:template>

<xsl:template match = "/" >

      <!-- Importation du module a tester -->
      <xsl:variable name="nom_module">
            <xsl:value-of        select="top/attributelist/attribute
[@name='module']/@value"/>
      </xsl:variable>
      <xsl:text>import </xsl:text>
      <xsl:value-of select="$nom_module"/>
      <xsl:text>&#xA;&#xA;</xsl:text>
```

```
      <!-- Ecriture des appels des fonctions -->
      <xsl:for-each select ="top/include/class" >
            <xsl:variable name="nom" select="attributelist/attribute
[@name='name']/@value"/>
            <xsl:text>#Creation de l'objet </xsl:text>
            <xsl:value-of select="$nom"/>
            <xsl:text>&#xA;</xsl:text>
            <xsl:value-of select="$nom"/>
            <xsl:text> = </xsl:text>
            <xsl:value-of select="$nom_module"/>
            <xsl:text>.</xsl:text>
            <xsl:value-of select="$nom"/>
            <xsl:text>()&#xA;</xsl:text>
            <xsl:for-each       select="cdecl/attributelist/attribute
[@name='name']">
                  <xsl:if  test="contains(../attribute[@name='decl']/
@value,'f(')">
                        <xsl:text># Appel de la fonction </xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:text>&#xA;</xsl:text>

                        <!-- Saisie des parametres -->
                        <xsl:for-each                          select=
"../parmlist/parm/attributelist" >
                              <xsl:call-template name="saisie_param">
                                    <xsl:with-param         name="nom"
select="attribute[@name='name']/@value"/>
                                    <xsl:with-param        name="type"
select="attribute[@name='type']/@value"/>
                                    <xsl:with-param       name="noeud"
select="current()"/>
                                    <xsl:with-param      name="module"
select="$nom_module"/>
                              </xsl:call-template>
                        </xsl:for-each>

                        <!-- Appel de la fonction -->

                        <xsl:if        test="not(contains(../attribute
[@name='type']/@value,'void'))">
                              <xsl:text>res = </xsl:text>
                        </xsl:if>
                        <xsl:value-of select="$nom"/>
                        <xsl:text>.</xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:text>(</xsl:text>
                        <xsl:for-each                          select=
"../parmlist/parm/attributelist" >
                              <xsl:value-of          select="attribute
[@name='name']/@value"/>
```

```
                        <xsl:if test="not(position()=last())">
                              <xsl:text> ,</xsl:text>
                        </xsl:if>
                  </xsl:for-each>
                  <xsl:text>)&#xA;</xsl:text>

                  <!-- Affichage de la sortie -->

                  <xsl:if        test="not(contains(../attribute
[@name='type']/@value,'void'))">
                        <xsl:text>print "Resultat de </xsl:text>
                        <xsl:value-of select="@value"/>
                        <xsl:text>"&#xA;</xsl:text>
                        <xsl:text>print </xsl:text>
                        <xsl:call-template name="retour_fct">
                              <xsl:with-param         name="type"
select="../attribute[@name='type']/@value"/>
                              <xsl:with-param
name="nom">res</xsl:with-param>
                              <xsl:with-param       name="module"
select="$nom_module"/>
                        </xsl:call-template>
                        <xsl:text>&#xA;</xsl:text>
                  </xsl:if>
            </xsl:if>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
   </xsl:for-each>
   <xsl:text>&#xA;</xsl:text>
</xsl:template>
</xsl:stylesheet>
```

7- Le document de spécification